Minimizing incomplete automata

Marie-Pierre Béal* Maxime Crochemore†

March 17, 2008

Abstract

We develop a $O(m \log n)$-time and $O(k+n+m)$-space algorithm for minimizing incomplete deterministic automata, where n is the number of states, m the number of edges, and k the size of the alphabet. Minimization reduces to the partial Functional coarsest partition problem. Our algorithm is a slight variant of Hopcroft's algorithm for minimizing deterministic complete automata.
Keywords: algorithms, automata, minimization, partitioning.

1 Introduction

It is well known that rational languages may be represented equivalently as rational expressions or as deterministic finite automata. Among them, there exists up to a renaming of states a unique minimal automaton accepting a given rational language (see for example [11]). This automaton is indeed the morphic image of any automaton accepting the language.

The description of a language with its minimal automaton is important when space considerations matter in implementations of applications, such as in Pattern Matching or in Coding Systems, for example.

There are several ways to get the minimal automaton of a language. If operations on languages are feasible, the method based on quotient languages produces it directly (see [11, 20]). However, partitioning is the basic strategy to minimize automata. It is also stated as the Functional coarsest partition problem, in which the question is to compute a coarsest partition of a finite set that is compatible both with marked elements and a finite set of functions defined on the set. Existing algorithms assume that functions are everywhere defined, or equivalently that the automaton is complete. If it is not, completion is a straightforward operation but requires additional space. The simplest efficient algorithms to minimize an automaton is by Moore and runs in time $O(kn^2)$ for a (deterministic) automaton with n states on an alphabet of size k. The fastest known solution by Hopcroft [10, 1] runs in time $O(kn \log n)$ (see also [9, 5, 15]). The design implement a so-called "smaller half" strategy. The complexity of the problem is still an open question but there is a family of automata on which Hopcroft's algorithm runs effectively in $\Theta(n \log n)$ [4].

*Université Paris-Est, Institut Gaspard-Monge, 77454 Marne-la-Vallée Cedex 2, France. beal@univ-mlv.fr

†King's College London, Strand, London WC2R 2LS, UK, and Université Paris-Est, Institut Gaspard-Monge, 77454 Marne-la-Vallée Cedex 2, France. maxime.crochemore@kcl.ac.uk

The minimal automaton of a language can also be built directly for some rare specific languages like the set of strings ending with a given string [8] or the set of suffixes of a string [7], for example. In this two cases the algorithms can even be implemented to run in linear time. But most often the minimization step has to be done after the construction of a deterministic automaton accepting the considered language, as for automata obtained from rational expressions (see [11]). Although the notion of a minimal automaton does not exist for non-deterministic automata, it is still possible to find equivalent automata with the minimum number of states (see [13, 12]), but the problem is known to be computationally hard [14].

It is rather simple to see that minimization can be achieved in linear time when the alphabet has only one letter ($k = 1$) [19]. The solution makes use of the particular structure of the automaton and of some string algorithms. But the solution does not seem to extend to larger alphabets.

Another known situation for which a linear-time algorithm exists is the case of finite languages [21] ($O(kn)$ running time reducible to $O(n)$ with the technique of the present paper) and the specific aforementioned languages.

Hopcroft's strategy with a slight modification also works for partitioning a graph or equivalently a non-deterministic automaton on a one-letter alphabet [6, 18] leading to $O(m \log n)$ running time algorithms. To do so, the "smaller half" technique is replaced by the "all but the largest" strategy as the one we use here.

To achieve an $O(m \log n)$ running time for minimizing local automata, we use in [2] a procedure for merging states having the same immediate future instead of Hopcroft's method that relies on state discrimination.

When a complete deterministic automaton is given, its size is $O(kn)$ and Hopcroft's algorithm adds a mere $\log n$ factor to minimize it. But when the automaton is incomplete, the actual size of the automaton assumed to contain no useless state is $O(m)$, where m its number of edges, which ranges from n to kn. Room for improvement is larger here than just dropping the log factor of Hopcroft's algorithm, which requires a complete automaton.

Another solution reaching the same running time as ours has been published recently by Valmari and Lehtinen [22]. Their solution uses the "smaller half" strategy as Hopcroft's algorithm does. But in addition to Hopcroft's method managing state partitions, they maintain a partition of transitions to have a direct access to edges labeled by a given letter and coming in a block splitter used later in the partitioning process. Both partitions are refined at the same time.

We develop in the rest an $O(m \log n)$-time and $O(k+n+m)$-space algorithm for minimizing incomplete deterministic automata. Our method uses three ingredients. The first one is the "all but the largest" strategy, a generalization of the "smaller half" strategy, which is used in [6]. In this strategy, a block can be split in more than two pieces and only the largest piece does not become a splitter. The second ingredient is the notion of signature of a state, which represents its outgoing labeled edges, notion that is used successfully to get a linear-time algorithm for minimizing acyclic automata [21] and local automata [2]. The third ingredient is "weak sorting" [17], which is used as well in [22], although they do not refer to it. We believe that our solution provides a simpler description of the whole algorithm.

2 Minimizing incomplete deterministic automata

2.1 Functional coarsest partition

Let $\mathcal{A} = (U, F, T)$ be a deterministic (incomplete) automaton over a finite alphabet A. The notation means that U is the set of states, F is the set of edges, T is the set of final states. No initial state is specified since it does not play any role in the minimization process. A *successful* path is a path ending in a final state. We assume that the automaton is *trim*, *i.e.*, each state is co-accessible: there is at least one successful path starting from this state. This assumption is important to guarantee the correctness of the algorithm. But we do not assume that the graph underlying the automaton is connected.

We denote by E the sequence of partial relations $(E_a)_{a \in A}$ defined by $(p, q) \in E_a$ if and only if (p, a, q) is an edge of $\mathcal{A}$. Indeed, all these relations are (partial) functions because the automaton is deterministic.

For any subset $S \subseteq U$, any letter $a \in A$, let

$$E_a(S) = \{q \mid \exists p \in S \ \ (p, a, q) \in F\},$$
$$E_a^{-1}(S) = \{p \mid \exists q \in S \ \ (p, a, q) \in F\}.$$

Let also $E(S) = \cup_{a \in A} E_a(S)$ and $E^{-1}(S) = \cup_{a \in A} E_a^{-1}(S)$. If p is a state, we shall write $E_a(p)$ and $E_a^{-1}(p)$ instead of $E_a(\{p\})$ and $E_a^{-1}(\{p\})$ respectively. If $E_a(p)$ is not empty and $B \subseteq U$, we shall write $E_a(p) \in B$ instead of $E_a(p) \subseteq B$ since $E_a(p)$ is a singleton.

If $B \subseteq U$ and $S \subseteq U$, B is said to be *stable* with respect to S if, for any $a \in A$, either $B \subseteq E_a^{-1}(S)$ or $B \cap E_a^{-1}(S) = \emptyset$. If P is a partition of U, P is *stable* with respect to S if all the blocks (classes) of P are stable with respect to S. A partition P is *stable* if it is stable with respect to each of its own blocks.

Let Q and R be two stable partitions finer than a partition P. The partition $Q \cup R$ obtained by merging the blocks of Q and R having a non-empty intersection is stable, coarser than Q and R, and finer than P. There is thus a coarsest stable refinement of P.

The *partial functional coarsest partition problem* is that of finding, for a sequence of partial functions $(E_a)_{a \in A}$ and an initial partition P over a set U, the coarsest stable refinement of P, *i.e.*, the partition which admits every stable partition as a refinement. The coarsest partition has the fewest blocks among all stable partitions. This partition defines the minimal automaton when the initial partition distinguishes final and non-final states.

In discussing time bounds for this problem, we let k denote the size of the alphabet A, n denote the size of set of states U, and m denote the size of E which is the sum of sizes of E_a, *i.e.*, the number of edges of the automaton $\mathcal{A}$.

The (complete) functional coarsest partition problem (when all E_a are complete functions) was solved by Hopcroft in time $O(kn \log n)$ [10] and space $O(k \times n)$ with a process known as the "smaller half strategy"(see for instance [3], [9], or [15] for an implementation with this time complexity). The relational coarsest partition problem (when E is a relation) was solved by Cardon and Crochemore [6] in time $O(m \log n)$ and space $O(n + m)$ (see also [18]). Another kind of partitioning, namely s-partitioning with a one-letter alphabet, has been designed in linear time by Paige, Tarjan, and Bonic [19].

We design a $O(m \log n)$-time and $O(k+n+m)$-space algorithm for the partial functional coarsest partition problem which is a slight variant of Hopcroft's

algorithm which works in the complete case. As a consequence, it is simpler than Paige and Tarjan's algorithm for the relational coarsest partition problem. Our algorithm provides a $O(m \log n)$-time and $O(k+n+m)$-space algorithm for minimizing an incomplete deterministic automaton.

2.2 Description of the algorithm

For any partition Q and subset $S \subseteq U$, we denote by $\text{SPLIT}(S, Q)$ the refinement of Q obtained by replacing each block $B \in Q$ by the blocks $(B_i)_{i \in \mathfrak{P}(A)}$ defined by $B_i = \{p \in B \mid \forall a \in i, E_a(p) \in S \text{ and } \forall a \notin i, E_a(p) \text{ is empty or } E_a(p) \notin S\}$. In this operation the set S is called the *splitter*.

Lemma 1. *Let S be the union of some blocks of a partition Q that is stable relative to S and let $B \subseteq S$ be a block of Q. Then $\text{SPLIT}(B, Q)$ is stable with respect to $S - B$.*

Proof. Indeed, let B_i be a block of $\text{SPLIT}(B, Q)$, where $i \in \mathcal{P}(A)$. The block B_i is stable with respect to B, *i.e.*, for any letter a, $B_i \subseteq E_a^{-1}(B)$ or $B_i \cap E_a^{-1}(B) = \emptyset$. If $B_i \subseteq E_a^{-1}(B)$, we have $B_i \cap E_a^{-1}(S-B) = \emptyset$ since E_a is a partial function. Let us now assume that $B_i \cap E_a^{-1}(B) = \emptyset$. Since B_i is stable with respect to S, we have either $B_i \cap E_a^{-1}(S) = \emptyset$, and hence $B_i \cap E_a^{-1}(S - B) = \emptyset$, or $B_i \subseteq E_a^{-1}(S)$. In the latter case, $B_i \subseteq E_a^{-1}(S) - E_a^{-1}(B)$ and hence $B_i \subseteq E_a^{-1}(S - B)$. Which shows that $\text{SPLIT}(B, Q)$ is stable with respect to $S - B$. □

The previous statement means that $\text{SPLIT}(B, Q) = \text{SPLIT}(S - B, Q)$. This is the idea of Hopcroft's algorithm for the functional coarsest partition problem, which is still valid in the case of partial functions. The two following algorithms refine an initial partition Q.

```
HOPCROFT(𝒜 = (U, F, T))
1  L ← (smallest of T and U \ T)
2  while L ≠ ()
3        do S ← extract first of L
4           for each a ∈ A
5                do Q ← 2SPLIT(S, Q, a)
6                   for each set P split into non empty sets P1 and P2
7                        do L ← L·(smallest of P1 and P2)
8  return Q
```

```
MINIMIZATION(𝒜 = (U, F, T))
1  L ← (T, U \ T)
2  while L ≠ ()
3        do S ← extract first of L
4           Q ← SPLIT(S, Q)
5           for each set P split into non empty blocks P1, P2, . . , Pr
6                do L ← L·(all but the largest of Pi)
7  return Q
```

The second algorithm maintains a queue L of possible splitters, initially containing every block of the initial partition P. The algorithm consists in initializing $Q = P$ and applying the following step until L is empty, at which time Q is the coarsest stable refinement:

Refine: Remove from L its first set S. Replace Q by $\text{SPLIT}(S, Q)$. Whenever a block $B \in Q$ splits into two or more nonempty blocks, add all but the largest to the back of L.

The correctness of this algorithm follows from the fact that if Q is stable with respect to a set S and P is split into smaller blocks $P_1, P_2, .. P_r$, then refining with respect to all but one of the sets P_i guarantees the stability with respect to the last one.

The property of being finer than the initial partition P is invariant during the computation. Let Q be the partition obtained at the final step and R be any other stable partition finer than P. Then $Q \cup R$ is a stable partition finer than P. Let Q_i be the finest partition computed during the process which is coarser than $Q \cup R$. If $Q \cup R \neq Q$, there is a block $B \in Q \cup R$ which is not stable under some block of Q_i, *i.e.*, under some union of blocks of $Q \cup R$. Thie implies that B is not stable under some block of $Q \cup R$, which contradicts the stability of $Q \cup R$. This proves that the algorithm computes the coarsest coarsest stable refinement of P.

To minimize an incomplete deterministic automaton $\mathcal{A} = (U, F, T)$, where U is the set of states, F the set of edges, and T the set of terminal states, we have to compute the *Nerode partition* of the set of states.

Let p be a state of $\mathcal{A}$. We denote by $F(p)$ the set of words u that are labels of a successful path starting from p. It is called the *future* of the state p. Two states p and q are said to be *Nerode equivalent* if and only if $F(p) = F(q)$. The *Nerode partition* is the partition induced by the Nerode equivalence. States of the minimal automaton are blocks of Nerode partition, edges and terminal states are defined accordingly.

The minimization algorithm is obtained by initializing the partition Q of states to $P = \{T, U - T\}$. The list L initially contains the two blocks of P.

Lemma 2. *The partition of states computed by the algorithm* Minimization *starting with $Q = \{T, U - T\}$ is the Nerode equivalence, that is, the coarsest stable refinement of $\{T, U - T\}$.*

Proof. The correctness of this algorithm follows from the fact that the Nerode partition N is a stable partition which is finer than the initial partition. Indeed, the Nerode partition is by definition finer than the initial partition $P = \{T, U - T\}$. Let us show that it is stable. Let us assume that there are blocks B, S of N and $p, q \in S$ such that $E_a(p) \in B$ and $E_a(q) \notin B$. Hence $E_a(p)$ and $E_a(q)$ are not Nerode equivalent and p and q neither. Let us now assume that $p, q \in S$ are such that $E_a(p) \in B$ and $E_a(q)$ is the empty set. Since $E_a(p)$ is co-accessible, p, q are not Nerode equivalent. Hence N is stable.

Conversely, we show that the Nerode partition is not finer than the coarsest stable refinement Q of $P = \{T, U - T\}$. Let us assume that it is false. Let p, q be two states in a same block B_0 of Q such that $F(p) \neq F(q)$. Then there is for instance a word u and a successfull path $(p_i, a_i, p_{i+1})_{0 \leq i \leq r-1}$ labelled by u starting from p such that there is no successfull path labelled by u starting from q. Let v be the longest prefix of u such that there is a path $(q_i, a_i, q_{i+1})_{0 \leq i \leq s-1}$ labelled by v starting from q. Since Q is stable, p_i, q_i for $0 \leq i \leq s$ belong to a same block B_i of Q. If $v = u$, then $p_r \in T$ and $q_r \notin T$ which contradicts the fact that Q is finer than $\{T, U - T\}$. If $v \neq u$, $s < r$ and B_s is not stable under the block of Q containing $E_{a_{s+1}}(p_s)$ which is contradiction. □

2.3 Description of the implementation

The implementation of the algorithm is analogue to Hopcroft's implementation of [3] (see also [18] or [16, Chapter 1]) despite the fact that the splittings of blocks become more complicated, because a block can be split into more than two pieces. Moreover, in order to get the claimed complexity, the splittings according to a block cannot be done sequentially for each letter of the alphabet. Most of this section is devoted to the description of the splitting procedure.

An efficient implementation requires several data structures. We represent each state $p \in U$ by a record (or structure) that we shall not distinguish from the state itself. We represent each block of a partition Q by a record that we shall not distinguish from the block itself. The blocks are numbered. We represent the partition Q by a doubly linked list of blocks. The edges (*i.e.*, the elements of all E_a) are records. The various records are linked together in the following way. Each state p points to an unordered list of the edges (p, a, q) (*i.e.*, the pairs $(p, q) \in E_a$ for some a), the list of the edges going out from p. Each state q points to an unordered list of the edges (p, a, q), the list of the edges coming in q. This allows scanning the set $E^{-1}(q)$ in time proportional to the number of its incoming edges. It also allows the scanning of all $E^{-1}(q)$, for all states q of a block B, in time proportional to the total size of the list of the edges coming in the block B. This scanning and the operations performed during it are described more precisely below.

Each block of Q has an associated integer giving its size and points to a doubly linked list of the elements in it. The double linking allows deletion in $O(1)$ time. Each state of U points to the unique block of Q containing it. The space needed for all the data structures is $O(n + m)$.

We now describe the splitting procedure under a splitter $B \in Q$. To each state p of $\cup_{q \in B} E^{-1}(q)$ is associated a set $set(p)$ whose elements are the letters a such that there is an edge labelled by a from p to a state in B. Each set has no duplicate elements. We denote by $set(p)^+$ the set $set(p)$ augmented by the block number of p. We then perform a set dicrimination of the states, *i.e.*, the states are discriminated according to their associated set $set(p)^+$. We use a technique called *weak sorting* in [17] which sorts all the sets according to an arbitrary total order computed by the algorithm. A weak sort can be seen as a weak form of a radix sort. It takes linear time in the sum of the number of elements in each set.

> PRODURE SPLIT(Q, B):
> Weak sort the states p in $\cup_{q \in B} E^{-1}(q)$ according to their associated set $set^+(p)$.

We now detail the steps of the weak sort. In a first step, we compute the set of letters a such that $E_a^{-1}(B)$ is not empty, and for each such letter, we compute the set of states p such that $set(p)$ contains a.

> STEP 1: BUCKET INITIALIZATION.
> During the scanning of the lists $E^{-1}(q)$ for all $q \in B$, we compute
>
> - an (unordered) linked list ℓ representing the set of letters a labels of some edge coming in B,
> - for each letter $a \in \ell$, the list $\ell(a)$ of states p in U such that $p \in E_a^{-1}(B)$.

This computation is done in time $O(|B| + \sum_{p\in B} |E^{-1}(p)|)$ and space $O(k)$ with the use of a table t of size k for which $t[a]$ points to the list $\ell(a)$. Each state p for which (p, a, q) is scanned is added to the bucket list $\ell(a)$.

In a second step, we compute the signatures the states contained in the above buckets. If p is a state contained in at least one list $\ell(a)$, its *signature* $\sigma(p)$ is the list of elements of *set*(p), *i.e.*, whose elements are the letters a for which there is an edge labelled by a from p to a state in B, in the order given by the list ℓ. Note that different signatures may have different lengths.

> Step 2: computation of the signatures.
> We scan the list ℓ and, for each $a \in \ell$, we scan the list $\ell(a)$, updating the signature of each state p. During this scanning we build an (unordered) linked list s of the states p belonging to at least one list $\ell(a)$.

This computation is done in time $O(\sum_{p\in B} |E^{-1}(p)|)$ and space $O(n)$ with the use of a table σ of size n such that $\sigma[p]$ points to the signature of p. Lists $\ell(a)$ are emptied after their scanning.

Note at this step that two states p, q belonging to a same block S before the splitting under B will belong to a same block after the splitting if and only if they have the same signature.

In a third step, we discriminate the states p contained in the above buckets with respect to their set *set*$^+(p)$. Two states having the same signature will be contiguous in the final list.

Recall that t is a table indexed by the letters of the alphabet. Let t_b be a table indexed by the possible block numbers (hence of size at most n). Let ℓ_1 be a linked list representing a set of block numbers and ℓ_2 be a linked list representing a set of letters. Both lists are without repetitions. Thus an element is added to the list only if it is not already contained in the list. The two lists are initially empty. We first discriminate with respect to the block number, and then with respect to the signatures. In order to simplify the presentation, we first assume that all signatures of the states in s have the same length r.

> Step 3: Discrimination of the states.
> For each state p in the list s do
>
> - let i be the the number of the block containing p,
> - add (in constant time) the number i to the list ℓ_1,
> - add the state p at the end of the list $t_b[i]$.
>
> The list s is then made empty and, for each block number i in the list ℓ_1, we concatenate the list $t_b[i]$ at the end of s.
>
> For j from 1 to r do
>
> - For each state p in the list s do
>   - add the state p at the end of the list $t[a]$, where a is the letter number j of the signature of p.
>   - add this letter a to ℓ_2.
> - Empty the list s and, for each letter a in the list ℓ_2, concatenate the list $t[a]$ at the end of s.

In the case where signatures are not of constant length, we first discriminate the states according to the length of their signature and apply the above procedure to each bucket. The discrimination is performed in time $O(\sum_{p \in B} |E^{-1}(p)|)$ and space $O(n+k)$. Note that the tables t, t_b do not need to be initialized by the use of sparse list implementation (see for instance [1] Exercise 2.12 p. 71).

At the end of the third step, the list s of states satisfies the property that two states belonging to a same block of Q and having the same signature are contiguous in s. The split blocks are obtained in the final Step 4.

> Step 4: Splitting of the blocks.
> We scan the list s obtained at the end of the third step and put two states belonging to a same block B of Q and having the same signature in a same split block B_i. States in the same block B_i are extracted from B. Sizes of B, B_i are updated during the extraction.

Step 4 requires a time $O(\sum_{p \in B} |E^{-1}(p)|)$ and no additional space.

Example 1. *We illustrate the weak sort. Let $\mathcal{A} = (U, F, T)$ be the automaton of Figure 1.*

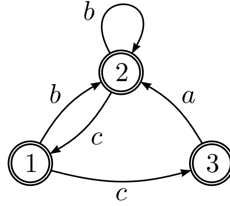

Figure 1: A deterministic incomplete automaton $\mathcal{A} = (U, F, T)$ over the alphabet $A = \{a, b, c\}$. All states are final.

Let $P = \{U\}$ be the initial partition of U. We detail the splitting under the block $U = \{1, 2, 3\}$. We successively get $\ell = (c, a, b)$,

$$\ell(a) = 3, \ \ell(b) = 1, 2, \ \ell(c) = 2, 1.$$

And the signatures are

$$\sigma(1) = cb, \ \sigma(2) = cb, \ \sigma(3) = a.$$

The initial list of states in step 3 is $s = (2, 1, 3)$. The final list obtained after the weak sort is $s = (2, 1, 3)$ and U is split into $U_1 = \{2, 1\}$ and $U_2 = \{3\}$.

The overall time complexity is obtained with classical arguments used for Hopcroft's algorithm. A given state $p \in U$ is in at most $\log_2 n$ different blocks B considered as splitters, since each successive such set is at most half the size of the previous one. Indeed, when a block S is split into sets S_i, all but the largest are becoming future splitters. Therefore we have described an implementation in which a refinement step with respect to a block B takes $O(|B| + \sum_{p \in B} |E^{-1}(p)|)$ time. From this a $O(m \log n)$ overall time bound on the algorithm follows by summing over all blocks B used for refinement and over all elements in such blocks. The space complexity is $O(k + n + m)$.

The conclusion lies in the next statement.

Theorem 3. *An incomplete automaton with n states, m edges, and k letters can be minimized in time $O(m \log n)$ and memory space $O(n + m + k)$.*

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[2] M.-P. Béal and M. Crochemore. Minimizing local automata. In *2007 IEEE International Symposium on Information Theory, ISIT 2007*, pages 1376–1380, 2007.

[3] D. Beauquier, J. Berstel, and P. Chrétienne. *Éléments d'algorithmique*. Masson, 1992.

[4] J. Berstel and O. Carton. On the complexity of Hopcroft's state minimization algorithm. In *Implementation and application of automata*, volume 3317 of *Lecture Notes in Comput. Sci.*, pages 35–44. Springer, Berlin, 2005.

[5] N. Blum. An $O(n \log n)$ implementation of the standard method for minimizing n-state finite automata. *Information Processing Letters*, 57:65–69, 1996.

[6] A. Cardon and M. Crochemore. Partitioning a graph in $O(A \log_2 V)$. *Theoret. Comput. Sci.*, 19(1):85–98, 1982.

[7] M. Crochemore. Transducers and repetitions. *Theoret. Comput. Sci.*, 45(1):63–86, 1986.

[8] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

[9] D. Gries. Describing an algorithm by hopcroft. *Acta Informatica*, 2:97–109, 1973.

[10] J. E. Hopcroft. An $n\log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi, editor, *Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971.

[11] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[12] L. Ilie, G. Navarro, and S. Yu. On NFA reductions. In *Theory is forever*, volume 3113 of *Lecture Notes in Comput. Sci.*, pages 112–124. Springer, Berlin, 2004.

[13] L. Ilie and S. Yu. Reducing NFAs by invariant equivalences. *Theoret. Comput. Sci.*, 306(1-3):373–390, 2003.

[14] T. Jiang and B. Ravikumar. Minimal NFA problems are hard. *SIAM J. Comput.*, 22(6):1117–1141, 1993.

[15] T. Knuutila. Re-describing an algorithm by hopcroft. *Theoret. Comput. Sci.*, 250(1-2):333–363, 2001.

[16] M. Lothaire. *Applied combinatorics on words*, volume 105 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, 2005.

[17] R. Paige. Efficient translation of external input in a dynamically typed language. In *Proc. 13th World Computer Congress*, volume 1, 1994.

[18] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

[19] R. Paige, R. E. Tarjan, and R. Bonic. A linear time solution to the single function coarsest partition problem. *Theoret. Comput. Sci.*, 40(1):67–84, 1985. Special issue: Eleventh international colloquium on automata, languages and programming (Antwerp, 1984).

[20] J.-E. Pin. *Varieties of formal languages*. Foundations of Computer Science. Plenum Publishing Corp., New York, 1986. With a preface by M.-P. Schützenberger, Translated from the French by A. Howie.

[21] D. Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoret. Comput. Sci.*, 92(1):181–189, 1992. Combinatorial Pattern Matching School (Paris, 1990).

[22] A. Valmari and P. Lehtinen. Efficient minimization of DFAs with partial transition functions. In S. Albers and P. Weil, editors, *Proc. 25th Annual Symposium on Theoretical Aspects of Computer Science*. ISBI Schloss Dagstuhl, 2008.